Edit Distance

CS 491 CAP

Dr. Mattox Beckman

University of Illinois at Urbana-Champaign
Department of Computer Science

Fall 2024

Objectives

- Use DP to determine the edit distance between two strings.

The Problem

- Given two strings *s* and *t*, how many "edits" does it take to tranform one to another?
    - Edit = insert, delete, or change.
    - Usually each of these "costs" one unit.
- Usually called the Levinstein Distance
- Examples:
    - changing `DATA` to `BETA` needs 2 steps.
    - changing `ETA` to `BETA` needs 1 step.
    - changing `GRETA` to `BETA` needs 2 steps.

Algorithm Outline

- Suppose you have strings `quiet` and `quaint`.
- Suppose You are comparing `i` in `quiet` to `a` in `quaint`.
    - We are assuming the beginnings have been edited.
    - What operations chould you do here?

The Näive Algorithm

Base Cases

```
// Thanks, Wikipedia!
int LD(string s, int len_s, string t, int len_t) {
  int cost;

  /* base case: empty strings */
  if (len_s == 0) return len_t;
  if (len_t == 0) return len_s;

  /* test if last characters of the strings match */
  if (s[len_s-1] == t[len_t-1])
      cost = 0;
  else
      cost = 1;
```

The Näive Algorithm, ctd

Recursive Case

```
  /* return minimum of delete char from s,
                       delete char from t,
                   and delete char from both */
  return minimum(LD(s, len_s - 1, t, len_t    ) + 1,
                 LD(s, len_s    , t, len_t - 1) + 1,
                 LD(s, len_s - 1, t, len_t - 1) + cost);
}
```

How can you convert this to DP?

You have to decide what is the state being remembered. . . .

Dynamic Programming using Memoization

Base Cases

```
int LD(const char *s, int len_s, const char *t, int len_t)
{
  vvi dp = vvi(len_s + 1, vi(len_t +1));
  int cost;

  for(int i=0; i<=len_s; ++i)
     dp[i][0] = i;

  for(int i=0; i<=len_t; ++i)
     dp[0][i] = i;
```

Dynamic Programming using Memoization, ctd

Memoized Part

```
  for(int i=1; i<=len_s; ++i)
    for(j=1; j<=len_t; ++j) {
       cost = s[i] == t[j] ? 0 : 1;

       dp[i][j] = minimum(dp[i-1][j] + 1,
                          dp[i][j-1] + 1,
                          dp[i-1][j-1] + cost);
       }
   return dp[len_s][len_t];
}
```